Freeform Search

Database: US Pre-Grant Publication Full-Text Database
US Patents Full-Text Database
US OCR Full-Text Database
EPO Abstracts Database
JPO Abstracts Database
Derwent World Patents Index
IBM Technical Disclosure Bulletins

Term:

Display: 10 **Documents in Display Format:** - **Starting with Number** 1

Generate: ○ **Hit List** ◉ **Hit Count** ○ **Side by Side** ○ **Image**

Search Clear Interrupt

Search History

DATE: Saturday, October 14, 2006 **Purge Queries** Printable Copy Create Case

Set Name side by side	Query	Hit Count	Set Name result set
DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=OR			
L30	l25 and (object-oriented or object adj oriented)	46	L30
L29	L25 and (relational or relation) near2 (database or data with base)	24	L29
DB=USPT; PLUR=YES; OP=OR			
L28	'6385612'.pn.	1	L28
DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=OR			
L27	L26 and (relational or relation) near2 (database or data with base)	19	L27
L26	L25 and objects	1440	L26
L25	(financial near2 service near2 organization or "fso")	2450	L25
DB=USPT; PLUR=YES; OP=OR			
L24	'5937189'.pn.	1	L24
L23	'5937189'.pn.	1	L23
L22	'6446086'.pn.	1	L22
DB=PGPB,USPT,USOC,EPAB,JPAB,DWPI,TDBD; PLUR=YES; OP=OR			
L21	L20 and (relational or relation) near2 (database or data with base)	10	L21
L20	L19 and organization	114	L20
L19	(financial near2 service near2 organization near2 model or "fso")	2264	L19

L18	L17 and (relational or relation) near2 (data with base or database)	23	L18
L17	L16 and objects	60	L17
L16	L15 and (organization or unit)	67	L16
L15	financial near2 service near2 model	75	L15
L14	711/217	730	L14
L13	711/216	555	L13
L12	711.clas.	31725	L12
L11	715.clas.	27062	L11
L10	715/533	119	L10
L9	715/513	2659	L9
L8	707/103r	2047	L8
L7	707/100	8861	L7
L6	707.clas.	38274	L6
L5	705.clas.	45321	L5
L4	705/44	1218	L4
L3	705/35	2681	L3
L2	705/5	1023	L2
L1	705/1	6320	L1

END OF SEARCH HISTORY

L18: Entry 21 of 23 File: PGPB Sep 27, 2001

PGPUB-DOCUMENT-NUMBER: 20010025264
PGPUB-FILING-TYPE: new
DOCUMENT-IDENTIFIER: US 20010025264 A1

TITLE: OBJECT ORIENTED SYSTEM FOR MANAGING COMPLEX FINANCIAL INSTRUMENTS

PUBLICATION-DATE: September 27, 2001

INVENTOR-INFORMATION:

NAME	CITY	STATE	COUNTRY
DEADDIO, MICHAEL	WHITE PLAINS	NY	US
KRAMER, AXEL	NEW YORK	NY	US

APPL-NO: 09/127341 [PALM]
DATE FILED: July 31, 1998

CONTINUED PROSECUTION APPLICATION: CPA

INT-CL-PUBLISHED: [07] G06F 17/60

INT-CL-CURRENT:

TYPE	IPC	DATE
CIPP	G06 Q 40/00	20060101

US-CL-PUBLISHED: 705/36; 705/10, 705/38
US-CL-CURRENT: 705/36R; 705/10, 705/38

REPRESENTATIVE-FIGURES: 1

ABSTRACT:

Object oriented design strategies and patterns are applied to financial data processing systems for processing and modeling of financial products (also referred to as financial instruments) with an emphasis being on derivative products. The system employs valuation independent, well-defined financial components (also referred to as financial events) that can be combined to build new financial structures. A general purpose software model is provided for representing the structure and characteristics of these products. A declarative specification language is provided to describe financial instruments in a consistent manner that lends itself to processing in such an object oriented system. A general traversal process is provided that can be applied to the macro structure of a financial instrument to implement various functions that produce results based on such information, such as the stream of financial events associated with the instrument, or the pricing or valuation of the instrument. Techniques including double dispatch and other mechanisms are further provided to provide flexible means of associating the appropriate processing methods with the diverse range of instrument

characteristics that are encountered in a typical financial institution's course of business.

Previous Doc Next Doc Go to Doc#

L18: Entry 21 of 23 File: PGPB Sep 27, 2001

DOCUMENT-IDENTIFIER: US 20010025264 A1
TITLE: OBJECT ORIENTED SYSTEM FOR MANAGING COMPLEX FINANCIAL INSTRUMENTS

Abstract Paragraph:
Object oriented design strategies and patterns are applied to financial data processing systems for processing and modeling of financial products (also referred to as financial instruments) with an emphasis being on derivative products. The system employs valuation independent, well-defined financial components (also referred to as financial events) that can be combined to build new financial structures. A general purpose software model is provided for representing the structure and characteristics of these products. A declarative specification language is provided to describe financial instruments in a consistent manner that lends itself to processing in such an object oriented system. A general traversal process is provided that can be applied to the macro structure of a financial instrument to implement various functions that produce results based on such information, such as the stream of financial events associated with the instrument, or the pricing or valuation of the instrument. Techniques including double dispatch and other mechanisms are further provided to provide flexible means of associating the appropriate processing methods with the diverse range of instrument characteristics that are encountered in a typical financial institution's course of business.

Summary of Invention Paragraph:
[0008] Much thought and effort has been put into the study of valuation/pricing models for the financial services industry. These studies are often of a highly theoretical and academic nature. While the continued development of pricing models is absolutely essential for the evolution of the financial business it does not supply the entire solution.

Summary of Invention Paragraph:
[0012] It is an object of the present invention to address the issues of consistency, manageability and modifiability described above by applying object oriented design strategies and patterns to the modeling and processing of financial products (also referred to as financial instruments) with an emphasis on derivative products. It is a further object of the invention to provide means to specify financial instruments in a consistent manner that lends itself to controlled and convenient instrument development and data entry. It is still a further object of the invention to provide a general means of processing financial data that is directed at the macro structure of a financial instrument but may be applied without variance for individual instrument characteristics.

Summary of Invention Paragraph:
[0013] To accomplish these and other objectives of the invention, we have developed a system that employs valuation independent, well-defined financial components (also referred to as financial events) that can be combined to build new financial structures. "Valuation independent" means that financial instruments are modeled independently of their valuation methodologies. A general purpose software model is provided and implemented in this system for representing the structure and characteristics of these products. A declarative specification language is provided to describe financial instruments in a consistent manner that lends itself to

processing in such an object oriented system. A general traversal process is provided that can be applied to the macro structure of a financial instrument to implement various functions that produce results based on such information, such as the stream of financial events associated with the instrument, or the pricing or valuation of the instrument. We also provide within this system a framework for processing the resulting structures in a way that supports various valuation methodologies. Techniques including double dispatch and other mechanisms are utilized to provide flexible means of associating the appropriate processing methods with the diverse range of instrument characteristics that are encountered in a typical financial institution's course of business. Such techniques allow developers to quickly and easily incorporate new valuation methodologies into the system as these methodologies become available.

Summary of Invention Paragraph:
[0015] This design has been implemented in Smalltalk and Java and we believe it can be implemented in any other object oriented language. Of course, since each development environment has its own tradeoffs, the implementation of such a complex system in a new development environment will present challenges, but ones we believe to be within the competence of persons reasonably skilled in the art.

Brief Description of Drawings Paragraph:
[0022] FIG. 6 is a block diagram showing the use of alternative instrument parameter instantiation for purposes of relational or object oriented persistent storage.

Brief Description of Drawings Paragraph:
[0023] FIG. 7 is a block diagram showing an overall view of the structure and processing of the present invention, including the static and event representation of financial instruments; the event extraction transformation process that generates the event representation from the static representation, and the stream processing objects that act upon the event representation to provide the desired business results and data.

Brief Description of Drawings Paragraph:
[0026] FIG. 10 is a block diagram showing an example of a class hierarchy that might be used to implement processing objects.

Detail Description Paragraph:
[0062] A financial instrument's object structure is illustrated in FIG. 4. The instrument instance holds onto an instance of each piece of its structure. It is important to recognize that every instrument, regardless of its type or specific details, has this structure. It is the specialization of the internal instances that differentiates one instrument from another.

Detail Description Paragraph:
[0075] There is a fixed interface for accessing and setting these parameters. As long as this interface is adhered to, parameter objects can be implemented in arbitrary ways and multiple implementations can coexist in harmony. The benefit of this flexibility is shown in the following example.

Detail Description Paragraph:
[0076] Consider the case in which a system is designed to integrate both a high volume "vanilla" financial business as well as a low volume, structured, "exotic" financial business. In most cases, a relational database technology is chosen for the vanilla business because it is characterized by relatively well defined and stable instrument structures. The exotic financial business, on the other hand, is characterized by the rapid development of highly complex instrument structures, therefore object oriented databases can be better suited. It is typically true that an exotics business uses associated vanilla financial products for hedging purposes. Therefore we would like to be able to store the vanilla products in

either database technology.

Detail Description Paragraph:
[0077] This problem can be easily solved with the instrument parameter paradigm. Essentially, there would be two hierarchies of instrument parameter objects; one that could be made persistent in a relational database and one that could be made persistent in an object oriented database. There would be a subclass in each hierarchy specifically for the given vanilla product. Both classes would support the same interface and would also support the same number and names of variables. Therefore, it would be transparent to the rest of the system, and to the financial instrument itself, what kind of database technology was being used. The actual kind of class needed would be determined by the underlying database requirements. It is also possible to switch, "on the fly", between database technologies as the different instances of parameter objects are essentially equivalent. This is illustrated in FIG. 6.

Detail Description Paragraph:
[0078] Another important advantage of this mechanism is that the class hierarchy of parameters is independent of the class hierarchy of all the other portions of the instrument. This separates the type of instrument from the state and behavior of the object that represents the parameters.

Detail Description Paragraph:
[0080] Financial components model simple, very narrow portions of real finance business concepts. These components must be reusable and very straightforward to understand and manipulate. When new financial instrument specifications are defined, developers must use pre-existing financial components unless they require a business concept that is not yet implemented. This achieves the composition strategy stated earlier because each business concept will be modeled by one, and only one, software object.

Detail Description Paragraph:
[0081] Using one software object to model a business concept allows for a very flexible system environment. Developers can expect the same object, and therefore the same behavior and state information, for a given business object in any financial instrument. Therefore, adding new types of instrument processing functionality can be accomplished quite generically. This concept will be detailed in Section 3 ("Generic Processing Framework") below.

Detail Description Paragraph:
[0082] If a development team adheres to this model one can expect the number of financial components for a given financial business environment to level off quickly. Essentially, the total number of business concepts is exhausted quite rapidly and therefore there is no need to develop more financial components. (For example, we found that about 45 basic components are sufficient to model all fixed income derivative instruments at JP Morgan.) Of course, this does not mean that the number of possible financial instruments is limited in any way. Since the instruments are built from organizations of different combinations of the components the permutations are, in effect, limitless. This organization is formalized via the instrument specification which will be detailed in the Section 2.3 ("Instrument Specification") below.

Detail Description Paragraph:
[0083] In general, the most basic business concepts become the base component class for all further specialization. These base classes typically identify and formalize the framework and interface for each business concept. This results in a component object hierarchy that is neatly and clearly divided along business lines. It allows developers to strongly leverage the inheritance tools available in all object oriented environments. The following example illustrates this strategy.

Detail Description Paragraph:
[0123] An important concept to note is that processing objects can only process the instrument in its Event Stream representation. This encapsulates the processing domain from the instrument specification domain. A processor should therefore be able to process an instrument's event stream regardless of the way in which it was specified.

Detail Description Paragraph:
[0139] Within the scope of the date interval iterator three financial components are defined; a rate event template, an accrual event template and a payment event template. Looking closely we can see how the variables for each instance are obtained from the instrument parameters, the iteration period object or an object created earlier in the process. We can tell from this declarative representation of the specification that each instantiated payment event would point directly to, and therefore contain, an accrual event in its accrual period. The accrual event would point directly to, and therefore contain, the rate event. We also find that the payment date is always 2 days after the end of the accrual period.

Detail Description Paragraph:
[0143] This specification has a similar structure as the interest rate swap leg specification but the rate template has been replace by an OptionRateTemplate that then refers to a RateTemplate. Here the developer is able to reuse the same building blocks for another financial instrument specification. For example, the same YearFractionAccrualTempla- te is used with a new type of rate template without any modification. This is allowed because the micro-structure of the accrual event specifies an interface for the rate variable that is supported by both the OptionRateTemplate and RateTemplate objects.

Detail Description Paragraph:
[0147] One very important rule in the event extraction process is that that the event structure is immutable with respect to time. The same process must always generate exactly the same event structure given the same parameter set and instrument specification, regardless of the relative system time. It is the responsibility of the processing object to ignore the events that are not applicable based on any specified date. This significantly simplifies the testing and verification of the event extraction process. The algorithmic event generation option that was described earlier has no explicit event extraction process as the code itself builds the events.

Detail Description Paragraph:
[0150] During the event extraction process, the processing of the date period iterator will create one instance of each component within its scope. This process also supplies each component with a period object that specifies the date interval of the current iteration. The processing of the iterator essentially functions as the date generation portion of the system.

Detail Description Paragraph:
[0157] The nature of this intermediate representation is an object graph of spec objects. For each financial component template, and for all other elements of the instrument description that hold the financial component templates together, there exists a spec object. This spec object holds the state of the financial template. The state of a financial template consists of either simple data types, like integers, floats, dates, etc., or references to variables or events which would be generated during the generic event extraction process. Examples for such references are: the last payment event, all reset events between two dates, etc. Since this is a description of the event structure, and not the actual event structure itself, such a reference can not be stated by direct reference, but must be stated implicitly via the events collected during the event extraction process.

Detail Description Paragraph:

[0158] All spec objects also implement a double dispatching protocol for the event extraction mechanism. Beyond that double dispatching method, spec objects do not define any interesting behavior. All the knowledge on how to extract events is contained in the event extraction mechanism.

Detail Description Paragraph:
[0159] For instruments described in FML this implies that an FML parser generates such an object graph. Standard computer science parsing technologies can be applied to generate such a parse tree. For instruments described with a programming language dependent specification, this implies that such a specification would generate this object graph using programming language specific statements, in essence creating spec objects and "plugging" them together in order to form the object graph.

Detail Description Paragraph:
[0160] The intermediate spec object graph needs to be constructed only once for each kind of instrument. It does not contain any information particular to a specific instrument. This information is provided externally, as the instrument parameters, as explained above.

Detail Description Paragraph:
[0161] The generic event extractor that generates the "canonical" event structure makes use of the instrument parameterization and the spec object graph for the instrument kind. It double dispatches with the spec objects in the graph and, in that process, creates the appropriate events, collects them, and resolves references described in the instrument specification. Note, this is very similar to the later processing of those financial events, where financial events do not know anything about the valuation method, and all that knowledge is contained in the event processing mechanism.

Detail Description Paragraph:
[0162] In the following we discuss the objects and concepts in the intermediate representation, and the generic event extraction process in detail.

Detail Description Paragraph:
[0163] 2.6.2. Intermediate Representation Objects

Detail Description Paragraph:
[0164] The objects contained in the intermediate representation fall into four categories: structural, parameterization, references, and events.

Detail Description Paragraph:
[0165] Classes and objects in the structural category describe the relationships of objects in the large: containment, grouping and naming, as well as periodicity. Classes in this category are NamedSpec, NestedSpec, SequenceSpec, and DatePeriods.

Detail Description Paragraph:
[0166] State classes describe which aspects of a spec can be parameterized, what the parameterized data for a particular instance looks like, how to represent temporary state, and how to represent the internal state of spec objects. Classes in this category are TempSpec, VarSpec, DataSpec, and FieldSpec.

Detail Description Paragraph:
[0167] Reference classes describe objects that are used to form the value of state objects. All non-literal references form the network of internal relationships between spec objects. Classes contained in this category are LiteralReference, NamedReference and Script-Reference.

Detail Description Paragraph:
[0169] An example for an intermediate representation is presented in Listing 5. The

specification from Listing 3 including a description of the required variables derived from Listing 1 are shown as intermediate objects. Indenting denotes containment.

Detail Description Paragraph:
[0177] The last interesting way to assign a value is via a ScriptReference. The "payDate" FieldSpec within the InterestPaymentSpec describes a value which is a ScriptReference containing: "period.end+days". This, in effect, is just a pretty print version of a more complex object tree containing a NamedReference and a LiteralReference, spelled out it reads: Script("addDays", Ref(period.end), LiteralReference(2)). This script is evaluated during processing and will result in the appropriate value (for all processors which actually create a value for this field in this event).

Detail Description Paragraph:
[0183] The intermediate representation can be processed in various ways. Analogous to the ideas about valuing financial instruments and the benefits of separating the description of a financial instrument from its processing, the processing of the intermediate representation is separated from the intermediate representation. There is no knowledge in the intermediate representation on how the processing is done, except a generic mechanism that enables the processor to know which object of the intermediate representation is to be processed: the double dispatching methods.

Detail Description Paragraph:
[0186] 2.6.4. Event Extraction Objects

Detail Description Paragraph:
[0190] The protocol defined in Spec, the interface all intermediate representation objects must adhere to, requires that each Spec understands the message traverseIn (SpecWalker w), see Listing 9. Each SpecWalker requires that each implementation of a SpecWalker understands traversal messages for all intermediate representation objects, as displayed in Listing 9.

Detail Description Paragraph:
[0193] The traversal of specs does not return any value. Thus references have their own double dispatch protocol which enables the SpecWalker to retrieve the value of a reference within the context of the current processing. A reference can be asked to getLiteralIn(aSpecWalker) and will bounce back to the SpecWalker with aSpecWalker.getLiteral(this), or in Smalltalk aSpecWalker getLiteralNamedReference: self. Returned is an object which represents the value of this reference at this point of processing. How the reference is resolved and how events are collected is discussed in the following section.

Detail Description Paragraph:
[0196] The ResolvingSpecWalker keeps an instance conforming to the WalkingContext interface (see Listing 10). This is an object which maintains a stack of Bindings and can be asked to push and pop a new Bindings object, to store and retrieve the value for a given name, and to collect and retrieve events.

Detail Description Paragraph:
[0197] Every time the resolving spec walker processes a structural intermediate representation object it pushed a new Bindings object on the context. If a NamedSpec is processed the name of the spec is pushed as well. This stack of bindings and names forms the basis for the value lookup mechanism implemented in the concrete implementation for the WalkingContext. Currently one such implementation exists, the ResolvingContext.

Detail Description Paragraph:
[0198] When an object is stored into the WalkingContext it is just stored into the

top Bindings object. When an object is retrieved via its name from the context and can not be found in the top binding, the subsequent bindings are searched. For each bindings frame which is named, the name is used as a prefix to the current search name. This implements the overwriting mechanism presented in Listing 7. Note, that the initial name will never be a compound name (using the dot-notation), since the access of names from outer scopes is not permitted.

Detail Description Paragraph:
[0200] A TempSpec and DataSpec are handled in similar ways, for a TempSpec the reference is retrieved and stored in the context, for a DataSpec, the literal object is simply stored into the context.

Detail Description Paragraph:
[0203] In contrast the EventExtractor can assume that all required parameters are defined. Instead, its focus is the creation of the appropriate Events and their collection into the proper event sequence. In contrast to the VariableExtractor it also needs to handle the iterator object properly.

Detail Description Paragraph:
[0205] After the state of the event is filled it is stored into the context under the sequence name for that EventSpec. When collecting events the ResolvingContext asks the Event for its preferred Event Collection object. Thus payment events can be collected into a payments collection, whereas reset events are collected into a resets collection. If no preference is given a generic collection is used. Which kind of collection is used is significant for the second step of processing the event structure and e.g. pricing it. The double dispatch mechanism there will make use of the different collections events have been collected in,

Detail Description Paragraph:
[0206] The iteration objects (e.g. DatesPeriodSpec) need to be rolled out when extracting events. Analogous to event generation, first a new DatePeriods object is created and its state is filled with values described in the collection of associated FieldSpecs. Based on that object an iterator object is initialized. A context for the iteration is pushed and for each iteration, the next value of the iterator is assigned into the name originally specified in the DatesPeriodSpec. Then the nested spec is traversed using the double dispatching protocol explained above. At the end the context is popped.

Detail Description Paragraph:
[0209] 2.7.1. Persistence--Relational vs. Object Oriented Databases

Detail Description Paragraph:
[0211] Modern object oriented databases allow complex object graphs to be made persistent. One can design the object model and assume that the database will map the object into a form of persistent storage. The object model effectively "is" the data model so no explicit data model is required. This greatly simplifies the entire development process from the point of view of the developer, especially when the object model becomes quite complex.

Detail Description Paragraph:
[0212] Relational databases, on the other hand, require not only an object model but an explicit data model and an explicit mapping between these two models. Both the models and the mapping strategy must be kept synchronized at all times for proper persistence behavior to be maintained. This requirement increases the amount of effort and overhead incurred by the business application developers when designing and implementing functionality.

Detail Description Paragraph:
[0213] The above discussion points out the major theoretical reasons why complex financial products can be better served by an object oriented database. There is

typically more overhead when using a relational database as a persistence mechanism for object oriented systems that depend upon highly complex object models.

Detail Description Paragraph:
[0214] As stated in Section 2 ("Financial Instrument Internal Structure"), every instance of an instrument can be represented either statically or as an event structure. This flexibility can allow a developer to employ a relational database with far more efficiency. It is now possible to make persistent only the static representation of an instance of any given instrument. The event structure can, by definition, always be obtained by transforming the static representation, thereby removing the need to make the event representation persistent.

Detail Description Paragraph:
[0223] As stated numerous times above, one of the key design goals was to separate the description of financial instruments from the processing of the instrument. Processing objects take the event representation of an instrument, i.e. its macro structure, and operate on the events and parameters to compute some output value or values. This was done to allow development of new instruments to proceed orthogonally to the development of new processing functionality.

Detail Description Paragraph:
[0224] It is very important to note that in this design the valuation operation, or pricing, is simply a specific type of processing. In terms of trading and risk management, pricing is the most critical operation. So it will typically be the most visible type of processing, but a type of processing nonetheless. This means that the general framework for both pricing and other kinds of processing objects will be exactly the same.

Detail Description Paragraph:
[0225] The design of this part of the system implements a framework that allows processing objects to walk over an instrument's financial event representation and implement an operation for each and every kind of event that it encounters. The concept is that the total processing algorithm is implemented via proper operation on all of the financial events. This is very much analogous to the concept described earlier; that financial instruments can be created by defining the structure and relationships of base financial components. That is, if we can define instruments via composition of basic components then we should also be able to define processing algorithms composed of operations on these basic components.

Detail Description Paragraph:
[0226] 3.1. Basic Object Oriented Processing Framework

Detail Description Paragraph:
[0227] The goal of this basic object oriented processing framework is to supply the tools necessary to implement event based processing for any purpose, be it pricing, processing, querying etc. This framework, once completed, is then leveraged to implement frameworks for the different types of functions. An example of a possible class hierarchy is depicted in FIG. 10:

Detail Description Paragraph:
[0253] Therefore, the actual methods and the order in which they are visited on the processor object is completely determined by the macro structure. In this way we say that the macro structure drives the processing of a financial instrument.

Detail Description Paragraph:
[0286] Therefore, the pricing framework must also provide for interfacing with analytical implementations that may not be in the same technology. This is typically done by providing an interface between the two technologies in which instrument data is passed into the analytics and pricing data is passed back to the main application. The cross-technology interface is simple to achieve. The issue

left to resolve is, how does one convert between the object model of the instrument in the main system and the representation required by the analytical implementation?

Detail Description Paragraph:
[0289] The classic object oriented view is that the definition of an object should encompass its state and its behavior. This design. On the other hand, specifically and explicitly separates what can be considered behavior from the description. The question raised is--Why is this deemed necessary?

Detail Description Paragraph:
[0291] Let us consider the classic case. Consider the instrument specification defined in Section 2.5 ("Simple Instrument Specification"), which describes a simplified version of an interest rate swap leg and an interest rate option instrument. In the classic object oriented paradigm, all processing methods would be implemented on these objects. For example, the swap object would know how to value itself, return a series of known and unknown payments etc.

Detail Description Paragraph:
[0292] To implement a properly modular system around instruments developed in this manner, developers typically define an appropriate interface. This interface typically represents all of the necessary functionality that any instrument must provide to the rest of the system to "fit in". The creation of a new instrument class simply requires that the developer implement the well-defined interface properly for the instrument in question. It is unimportant how the developer implements the interface for the instrument as the interface is meant to hide all the internal complexity of an instrument from other objects.

Detail Description Paragraph:
[0293] This strategy seems quite reasonable on first consideration. It is a properly "object oriented" approach and simple to implement. In practice, this approach would work quite well during the initial phases of a financial instrument model.

Detail Description Paragraph:
[0298] There is another reason, more specific to valuation models in the financial arena, for the separation of processing objects from instrument description objects.

Detail Description Paragraph:
[0301] This means that one instrument can be valued in several different methodologies and the decision as to which methodology is to be used can vary over time. The ability to easily choose a pricer object that implements the desired methodology is a very important and powerful advantage of this model. This is not easily done if all of the processing/pricing knowledge is implemented on the instrument itself.

Detail Description Paragraph:
[0302] An additional benefit is that pricing objects can share basic behavior via inheritance and that these objects can be used for multiple instruments. Otherwise, if the processing behavior is implemented directly on the instruments there would be a large degree of code duplication.

Detail Description Paragraph:
[0308] One example for this approach is an audit trail. Up to the point where a trade is confirmed no audit trail information needs to be kept, the instrument is in the front-office experimental stage. As soon as the trade is confirmed an audit spec and audit parameters are added to the instrument, stating which finance parameters can be changed at all, by who, and also how and where to record the audit trail. The audit trail might actually be recorded right in the instrument

itself, as read-only state related to the audit specification, or it might be recorded in a relational database the audit spec refers to.

Detail Description Paragraph:
[0323] Our experience shows that the model described here offers significant benefits to the developers of financial systems that must use object oriented technologies to represent financial instruments. As currently implemented, it is biased towards systems where pricing and large scale life cycle processing are the main deliverables. But, we believe that it is flexible and generic enough to be used successfully in various other capacities.

Detail Description Paragraph:
[0327] It is apparent from the foregoing that a new system has been developed that accomplishes the stated objects of the invention. While the presently existing embodiment and certain variations thereon have been described in detail, it will be apparent to those skilled in the art that the principles of the invention are readily adaptable to other adaptations and configurations of the systems described herein without departing from the scope and spirit of the invention, as defined in the following claims.

Detail Description Table CWU:
9 Listing 9 - Spec,'Reference and SpecWalker interface public interface Spec { public void traverseIn(SpecWalker s); } public interface Reference{ public Object getLiteralIn(Specwalker w) throws ReferenceNotDefined; } public interface Specwalker(void traverse(NamedSpec s); void traverse(sequencespec s); void traverse(DatePeriodsSpec s); void traverse(EventSpec s); void traverse(VarSpec s); void traverse(TempSpec s); void traverse(FieldState s); void traverse(DataSpec s); Object getLiteral (NamedReference ref) throws ReferenceNotDefined; Object getLiteral (ScriptReference ref) throws ReferenceNotDefined; Object getLiteral (LiteralReference ref) throws ReferenceNotDefined;)

Detail Description Table CWU:
10 Listing 10 - Bindings and WalkingContext interface public interface Bindings { public static Object UnknownValue = new Object(); public static Object InitialValue = new Object(); public void put(String name, Object value); public Object get(String name); } public interface WalkingContext{ public void push(String name, Bindings frame); public void push)Bindings frame); public void pop(); public Bindings peekBindings(); public String peekNames(); public void put(String name, Object value); public Object get(String name) throws ReferenceNotDefined; public String getScopeName(); public Events getEvents (String name) throws ReferenceNotDefined; public Hashtable getEvents(); public void collectEvent(String name, Event event); }

Detail Description Table CWU:
11 Listing 11 - Getting an object in the ResolvingContext public Object get (String n) throws ReferenceNotDefined{ Bindings frame; Object result; String name = n; String frameName; for (int i = frames.size()-1; i>=0; i--) { frame = (Bindings) frames.elementAt(i); result = frame.get(name); if (result != Bindings.UnknownValue) return result; frameName = (String)names.elementAt(i); if (frameName != UnnamedFrame) { name = frameName + ".multidot." + name; } } throw new ReferenceNotDefined(); }

Detail Description Table CWU:
12 Listing 12 - Traversing a VarSpec in the ResolvingSpecWalker public void traverse (VarSpec v) { Object value = null; Reference ref = new NamedReference (v.getName()); try{ value = ref.getLiteralIn(this); }catch(ReferenceNotDefined e) { System.out.printIn("Error: variable" + v.getName() + "not defined"); { context.put(v.getName(), value); {

Detail Description Table CWU:

13 Listing 13 - Traversing a VarSpec in the VariableExtractor public void traverse (VarSpec v) { Object value = null; Type type = null; String n = v.getName(); Reference ref = new NamedReference(n); try{ value = ref.getLiteralIn(this); }catch (ReferenceNotDefined e) { // register in variables String scope = context.getScopeName(); value = Bindings.InitialValue; type = v.getVarType(); String k; if (scope.equals ("")) { k = v.getName(); }else{ k = scope + ".multidot." + n; } variables.put(k, value, type); } context.put(n, value); }

CLAIMS:

7. A process implemented within means for processing financial data that transforms static representation of a financial instrument as created in accordance with claim 6 into a timeline of inter-related event objects that is specific to the given static representation.

8. The process of claim 7, wherein said timeline of inter-related event objects is composed of basic financial building blocks, known as "financial events or components", and constitutes in its entirety the financial event structure or macro structure of that particular financial instrument.

Previous Doc Next Doc Go to Doc#